

# JINSPIRED Developer

Software Metering, Metric Monitoring and Event Signaling for Java Developers

[Builds](#) [Usage](#) [Docs](#) [Repo](#) [API](#) [Install](#) [Config](#) [Console](#) [Site](#)

## The Good and B(AD) of Application Performance Management Measurement

by Architect on September 15, 2011

Application performance management vendors have long struggled with how to manage overhead in the instrumentation, measurement and collection of data. Over the years I've seen many application performance management vendors come and go each claiming to have made a breakthrough but yet each one not doing anything truly different other than shifting cost around, masking overhead and in nearly all cases collecting less data including what is truly [relevant](#).

There are two main approaches to application performance measurement (monitoring): code instrumentation (*event based*) and call stack (*statistical*) sampling. Every now and again the popularity of one approach over the other reverses and we get a slew of new "next generation" vendors claiming sampling solves all the problems of event based measurement, in particular high overhead, which is then followed by a period of disillusionment with the anemic and inaccurate data sets produced by such sample based solutions wherein we see the rise of the "next generation" event based vendors – again.

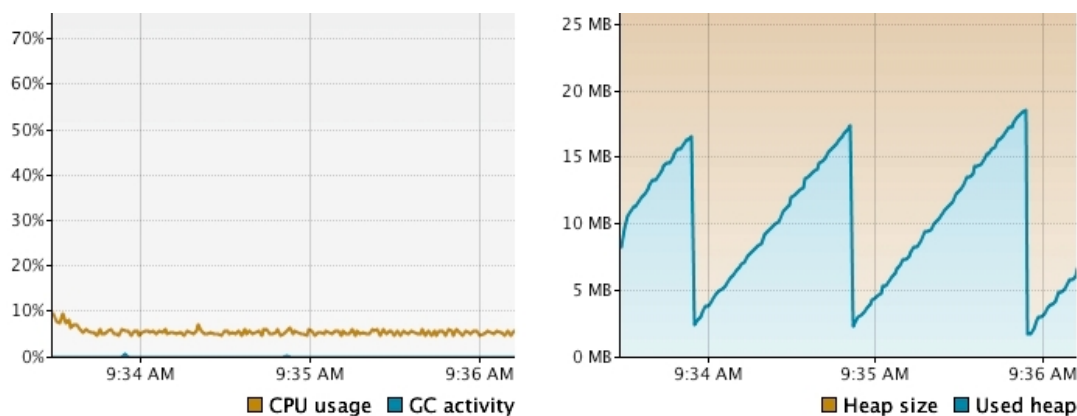
In all benchmarks [Intelligent Activity Metering](#) has proven to be hugely superior in every aspect. So much so that I really don't see the point of sampling. At this stage you are probably thinking "here we go he's going to give me the event based vendor pitch". But the fact of the matter is that software activity resource metering is event based in so far as instrumentation but not measurement and collection. Its smart which is something that can't be said of the solutions in both camps today.

In the first section of this blog I would like to focus on one of my pet peeves with ALL performance load testing tools – the load generated never creates a production like resource consumption profile largely invalidating findings or any comfort felt in a successful execution run. I have lost count of the times I've been called into a production site tasked with investigating why performance testing did not uncover issues that had been identified by our metering and metrics runtime **in production** only to be given two completely different resource (*metric*) profiles from production and test. My ideal performance testing tool is one that does not report a success until it has replicated (*maybe on a smaller scale but still apportioned correctly*) the same behavioral and resource consumption characteristics. Lets see can we apply this same criteria to application performance management solutions.

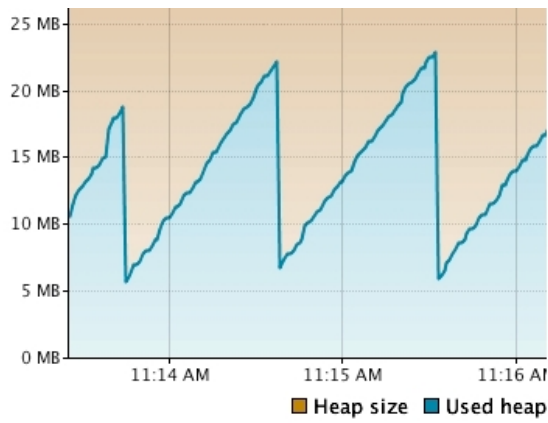
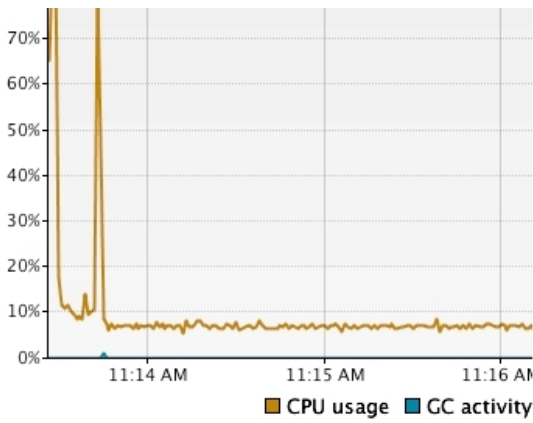
At the bottom of this blog you will find a code listing of a Java class I have used to simulate (*more so model*) the typical execution flow of request processing done by web applications and web containers today. The code creates 50 threads. Each thread executes an entry point call, `Client.iterate()`, which in turn makes a call to `$0()` which calls `$1()` which calls `$2()` and so on until eventually it arrives at the `$100()` method where it sleeps for a number of milliseconds with the sleep representing a typical slow remote call to a database, web service or messaging system. Along the way at each `$*()` method we call to `l()` and `r()` to simulate extraneous work performed at each processing layer.

I ran 3 tests. The first test with no instrumentation. The second with OpenCore's intelligent activity metering runtime. The third with AD's next generation "business transaction" (*business like method renaming*) solution which basically consists of using an event based measurement at the very top entry point then performing call stack sampling once the "business transaction" exceeds some threshold - *generally far too late and of little value and meaning due to the obvious disconnect with an incomplete of data set but we will come back to that later*. Again this is not a benchmark. Its to see whether things remain the same or not. Does the application performance management solution create an significant and unwanted behavioral and resource consumption profile change?

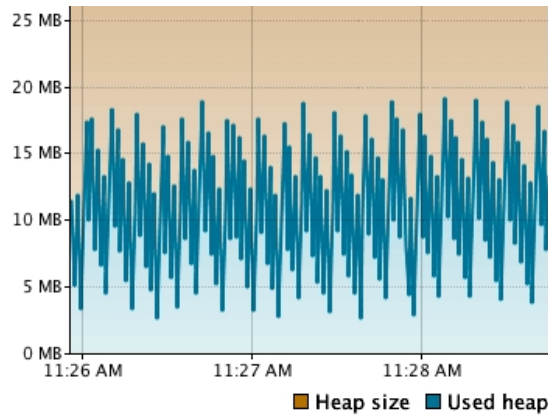
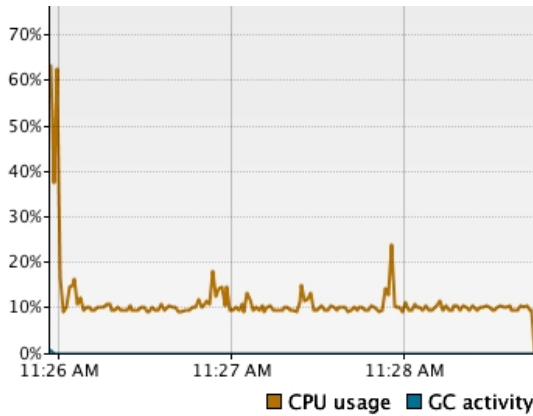
Here is the baseline resource (*cpu, heap*) consumption profile collected using an independent JMX based monitoring tool.



Below is the JXInsight/OpenCore resource consumption profile. The heap chart is relatively unchanged from the baseline. The cpu chart has two spikes not present in the baseline. The first sharp short cpu spike is caused by the instrumentation of the class bytecode loaded and the initialization of the metering runtime along with its pre-allocated data structures. The second spike is the metering hotspot strategy analysis kicking in disabling non-hotspot methods from ever been metered again which then results in the JVM dynamic runtime compiler optimizing away the resulting dead code. This is apparent in the drop in the CPU following the spike. Other than that everything seems pretty much the same with probably a tiny bit (<1%) higher cpu consumption.



Here is the resource consumption profile for AD for the exact same code as before. The heap profile has changed drastically. The cpu consumption likewise being double the baseline rate with some occasional mid point spikes. This really is a different beast altogether that is being measured caused more than likely by excessive object allocation in the creation of AD's "business transaction" context and the sampling of the call stack once the transaction has exceeded a threshold which was set at 10ms.



Lets now move onto the actual data collected and assessing its value and presentation quality - the achilles heal of call sampling especially one based on an initial threshold delay.

Below is the OpenCore probes metering model collected following completion of the test using the following settings added to a `jxinsight.override.config` file. For each method we have the *Count*, *Total*, *Inherent (Self) Total* and so on including *Min*, *Min (I)*, *Max*, and *Max (I)* not displayed here. Only one resource meter, `clock.time`, was included though OpenCore supports an unlimited number of built-in or custom ones. You'll notice that many of the probes have been disabled (grey circle) because of the hotspot analysis which occurs in steps of 1,000 by default. The `pojo.Client.sleep` probe is classified as the single hotspot which is correct in terms of the strategy settings.

```
jxinsight.server.probes.console.enabled=false
jxinsight.server.probes.aggregates.enabled=false
jxinsight.server.probes.strategy.hotspot.statistic=iavg
```

Metering Table							
Context	Probe	Meter	Count	Total	Total (I)	Avg	Avg (I)
⌘ 1515	🔵 pojo.Client.run	🕒 clock.time	500	*8,495,518,964	14,251,318	16,991,037	28,502
⌘ 1515	🟡 pojo.Client.sleep	🕒 clock.time	*1,500,000	8,480,794,686	*8,480,794,686	5,653	5,653
⌘ 1515	🔵 pojo.Client.main	🕒 clock.time	1	170,441,574	64	*170,441,574	64
⌘ 1515	🔵 pojo.Client.test	🕒 clock.time	10	170,441,510	170,441,212	17,044,151	*17,044,121
⌘ 1515	🔵 pojo.Client.iterate	🕒 clock.time	1,050	19,544,288	1,957	18,613	1
⌘ 1515	🟡 pojo.Client.\$0	🕒 clock.time	1,050	19,542,331	4,410	18,611	4
⌘ 1515	🟡 pojo.Client.\$1	🕒 clock.time	1,050	19,537,856	2,745	18,607	2
⌘ 1515	🟡 pojo.Client.\$2	🕒 clock.time	1,050	19,535,053	2,400	18,604	2
⌘ 1515	🟡 pojo.Client.\$3	🕒 clock.time	1,050	19,532,594	5,443	18,602	5
⌘ 1515	🟡 pojo.Client.\$4	🕒 clock.time	1,050	19,527,087	2,974	18,597	2
⌘ 1515	🟡 pojo.Client.\$5	🕒 clock.time	1,050	19,524,055	2,278	18,594	2
⌘ 1515	🟡 pojo.Client.\$6	🕒 clock.time	1,050	19,521,717	2,458	18,592	2
⌘ 1515	🟡 pojo.Client.\$7	🕒 clock.time	1,050	19,519,200	4,098	18,589	3
⌘ 1515	🟡 pojo.Client.\$8	🕒 clock.time	1,050	19,515,044	2,304	18,585	2
⌘ 1515	🟡 pojo.Client.\$9	🕒 clock.time	1,050	19,512,682	2,521	18,583	2
⌘ 1515	🟡 pojo.Client.\$10	🕒 clock.time	1,050	19,510,106	2,372	18,581	2
⌘ 1515	🟡 pojo.Client.\$11	🕒 clock.time	1,050	19,507,675	2,254	18,578	2
⌘ 1515	🟡 pojo.Client.\$12	🕒 clock.time	1,050	19,505,355	6,617	18,576	6
⌘ 1515	🟡 pojo.Client.\$13	🕒 clock.time	1,050	19,498,677	2,449	18,570	2

But what about the `pogo.Client.iterate()` method which in fact was the entry point for the "business transaction" as defined in the corresponding AD test. Because it's inherent avg was below 100 microseconds it got disabled but we can fix that so that all "business transaction" points are not enhanced by the strategy probes provider with the following setting.

```
jxinsight.server.probes.strategy.filter.enabled=true
jxinsight.server.probes.strategy.filter.exclude.name.groups=pojo.Client.iterate
```

Metering Table							
Context	Probe	Meter	Count	Total	Total (I)	Avg	Avg (I)
1515	pojo.Client.run	clock.time	500	*8,553,976,418	488,425	17,107,952	976
1515	pojo.Client.iterate	clock.time	500,000	8,553,487,993	14,744,410	17,106	29
1515	pojo.Client.sleep	clock.time	*1,500,000	8,538,089,495	*8,538,089,495	5,692	5,692
1515	pojo.Client.main	clock.time	1	171,743,525	114	*171,743,525	114
1515	pojo.Client.test	clock.time	10	171,743,411	171,743,098	17,174,341	*17,174,309
1515	pojo.Client.\$0	clock.time	1,050	19,647,363	11,056	18,711	10
1515	pojo.Client.\$1	clock.time	1,050	19,636,243	3,910	18,701	3
1515	pojo.Client.\$2	clock.time	1,050	19,632,271	4,526	18,697	4
1515	pojo.Client.\$3	clock.time	1,050	19,627,684	3,960	18,693	3
1515	pojo.Client.\$4	clock.time	1,050	19,623,659	4,816	18,689	4
1515	pojo.Client.\$5	clock.time	1,050	19,618,786	5,888	18,684	5
1515	pojo.Client.\$6	clock.time	1,050	19,612,833	5,026	18,678	4
1515	pojo.Client.\$7	clock.time	1,050	19,607,747	4,214	18,674	4
1515	pojo.Client.\$8	clock.time	1,050	19,603,472	4,659	18,669	4
1515	pojo.Client.\$9	clock.time	1,050	19,598,752	5,252	18,665	5
1515	pojo.Client.\$10	clock.time	1,050	19,593,442	3,928	18,660	3
1515	pojo.Client.\$11	clock.time	1,050	19,589,456	4,441	18,656	4
1515	pojo.Client.\$12	clock.time	1,050	19,584,950	4,012	18,652	3
1515	pojo.Client.\$13	clock.time	1,050	19,580,872	3,815	18,648	3
1515	pojo.Client.\$14	clock.time	1,050	19,576,994	8,930	18,644	8
1515	pojo.Client.\$15	clock.time	1,050	19,568,003	4,502	18,636	4
1515	pojo.Client.\$16	clock.time	1,050	19,563,439	4,981	18,631	4
1515	pojo.Client.\$17	clock.time	1,050	19,558,394	4,866	18,627	4

A call tree can be collected by enabling the tracking probes provider. Note how the metering tracking (call) tree shows `iterate()` calling directly `sleep()` because of the dynamic disablement of the other intervening probed calls. This is extremely useful since disabled probes can be hidden within the management console leaving the true essence of the performance behavior.

```
jxinsight.server.probes.tracking.enabled=true
```

Tracking Tree					
Context	Track	Meter	Count	Total	Avg
1515	<>	clock.time	501	*8,639,355,090	17,244,221
1515	pojo.Client.run	clock.time	500	8,469,588,543	16,939,177
1515	iterate	clock.time	500,000	8,468,947,658	16,937
1515	sleep	clock.time	*1,496,850	8,435,748,309	5,635
1515	\$0	clock.time	1,050	18,743,511	17,850
1515	\$1	clock.time	1,050	18,740,210	17,847
1515	\$2	clock.time	1,050	18,736,709	17,844
1515	\$3	clock.time	1,050	18,733,527	17,841
1515	\$4	clock.time	1,050	18,730,708	17,838
1515	\$5	clock.time	1,050	18,725,569	17,833
1515	\$6	clock.time	1,050	18,722,472	17,830
1515	\$7	clock.time	1,050	18,719,496	17,828
1515	l	clock.time	11	45	4
1515	r	clock.time	7	29	4
1515	l	clock.time	11	43	3
1515	r	clock.time	7	30	4
1515	l	clock.time	11	44	4
1515	r	clock.time	7	32	4
1515	l	clock.time	11	46	4
1515	r	clock.time	7	31	4
1515	l	clock.time	11	48	4
1515	r	clock.time	7	31	4
1515	l	clock.time	11	48	4
1515	r	clock.time	7	30	4
1515	l	clock.time	11	52	4
1515	r	clock.time	7	33	4
1515	pojo.Client.main	clock.time	1	169,766,547	*169,766,547

AD does not consolidate hierarchical data at the thread or process or cluster level but instead records individual transactions a little bit similar but less powerful than OpenCore's transaction probes provider. Here is an example of a transaction recorded by AD. Incomplete. Inaccurate. Irrelevant. No Count. No Max. No Min. Practically nothing is recorded at each of the call points other than what is provided by stack traces.

Name	Time (ms)
java.lang.Thread:run:680	0 ms (self) 0 %
pojo.Client:run:43	0 ms (self) 0 %
pojo.Client:iterate:48	7 ms (self) 36.8 %
pojo.Client:\$0:61	0 ms (self) 0 %

Here is another AD transaction recorded that was deemed a slow request. There is simply no data other than we know it exceeded its threshold. This is because AD waits until the threshold has already been exceeded and then begins call sampling and sleep cycle. You have no prior context and none after. Useless.

Name	Time (ms)
java.lang.Thread:run:680	0 ms (self) 0 %
pojo.Client:run:43	0 ms (self) 0 %
pojo.Client:iterate:48	0 ms (self) 0 %
pojo.Client:\$0:61	0 ms (self) 0 %

AD does not have to wait for a threshold to be exceeded. It offers the ability to sample every 100th transaction irrespective. Here is the complete data collected for such a sampled transaction. Nothing other than the entry point method has been measured or sampled.

Name	Time (ms)
pojo.Client:iterate	17 ms (self) 100 %

#### Code Listing

```
package pojo;

import static java.lang.System.nanoTime;
import static java.text.MessageFormat.format;

public class Client implements Runnable {

    private static final int THREAD_COUNT = 50;
    private static final int TEST_COUNT = 10;
    private static final int RUN_COUNT = 1000;
    private static final int SLEEP = 5;

    public static void main(String[] args) throws Throwable {

        for (int i = TEST_COUNT; i > 0; i--)
            test();

    }

    private static void test() throws Throwable {

        final Thread[] threads = new Thread[THREAD_COUNT];

        for (int i = 0; i < THREAD_COUNT; i++)
            threads[i] = new Thread(new Client());

        long start =.nanoTime();

        for (int i = 0; i < THREAD_COUNT; i++)
            threads[i].start();

        for (int i = 0; i < THREAD_COUNT; i++)
            threads[i].join();

        long end =.nanoTime();
        System.out.println(format("clocktime={0}", (end - start)));

    }

    public void run() {

        for (int i = RUN_COUNT; i > 0; i--)
            iterate();

    }

    private void iterate() { $0(); }

    private void l() { nanoTime(); }
    private void r() { nanoTime(); }

    private void $0() { l(); $1(); r(); }
    private void $1() { l(); $2(); r(); }

}
```

The following method mimics a slow exit (*hanging*) point

```
private void $98() { l(); $99(); r(); }
private void $99() { l(); sleep(); r(); }

private static void sleep() {
    try {
        Thread.sleep(SLEEP);
    } catch (InterruptedException e) { /** .. **/ }
}
```